Boss/Worker Model for Multi-GPU Programming

Stefano Pedemonte and Sebastien Ourselin

The Centre for Medial Image Computing, UCL, London, United Kingdom.

Abstract. The Boss/Worker (B/W) model provides a simple and effective abstraction for parallel execution of problems that present a high degree of task concurrency. This is achieved by providing automatic load balancing and by efficiently hiding the communication latencies. We address the issue of integrating general purpose graphics processing units (GPGPU) into a cluster by designing a novel Boss/Worker model which enables to tailor the distribution of workload across a non-uniform cluster, where some of the nodes, but not necessarily all, are equipped with GPU. Our solution particularly tailors the lack of awareness of GPU acceleration by the underlying messaging system and the selection and allocation of GPU devices. We present in detail our Boss/Worker design and an implementation based on the MPI messaging system. Finally, we demonstrate the scalability and performance of our approach applying the Boss/Worker programming model to an algorithm for Emission Tomographic Reconstruction.

Keywords: GPGPU, boss/worker, parallel programming, cloud computing, medical image computing, MPI, Python

1 Introduction

While the standardization of programming interfaces is allowing a number of applications to be ported to GPGPU, benefiting a 10 to 100 fold increase in performance when compared with single CPU implementations [1][2][3], scaling applications to run on multiple GPGPUs relies on ad-hoc solutions. Multithreading and multiprocessing technologies allow one to orchestrate the use of multiple graphics cards on the same machine, but the number of GPUs is limited by PCI bandwidth and physical size of the motherboard. Message passing technologies on the other hand allow one to scale applications to use multiple CPUs in homogeneous and heterogeneous clusters but there is scarcity of programming models and software support for multi-GPU systems [4][5].

Though it is possible to adapt communication protocols designed for parallel computing on distributed memory systems [6], such as MPI, to make use of GPUs, the complexity of the multiprocessor based GPU programming model and the additional complexity of the shared memory programming model hinder the design and development of applications. In this paper we investigate how the

Boss/Worker model can be adapted to distribute the execution of code over multiple GPUs in a distributed memory system.

The B/W model provides an abstraction over the communication protocol, guaranteeing automatic load balancing and a degree of latency hiding. Hiding the communication protocol, the B/W paradigm allows the programmer to be agnostic about the underlying mechanism that allows the nodes to exchange data and to ignore communication latency hiding and load balancing amongst the units of execution.
The B/W programming paradigm provides an abstraction over the communication protocol and guarantees automatic load balancing and latency hiding for computational problems that present a high degree of task parallelism [7][8]. Many such problems are encountered in the processing of imaging data.

We aim at providing a mechanism that allows us to distribute the code that was conceived for execution on single GPU over multiple GPUs with little additional effort. Furthermore we aim at providing a design of the B/W model that allows us to scale the distribution of tasks on a large range of platforms: multiple GPUs on a single machine, one or multiple GPUs per machine in a cluster, or a number of CPUs in a cluster or in a grid, achieving abstraction from the hardware and the communication systems.

We analyse the applicability of the B/W model to a distributed memory multi-GPU system and present an implementation of the B/W model that provides full abstraction over the network and messaging system and tailors the problems of detection and allocation of the GPU devices. and collaboration of tasks on a number of nodes.

In the first section we describe the B/W programming model and issues specific to GPU embedding; we then describe the implementation of the back-end based on MPI-2 and finally we validate performance and scalability on a genuine application for Emission Tomographic Reconstruction.

2 GPU aware Boss/Worker architecture

In order to scale execution of a program that makes use of the B/W abstraction on a number of different architectures, ranging from a single multi-core machine, to a GPU cluster, to a computing grid, we designed an API that builds an abstraction over the mechanism for data exchange. The back-end for communication and data exchange is an interchangeable module which can be reimplemented in order to enable the same applications to run under different computation environments.

In order to maintain a common API, it is necessary to address the issue of selection and allocation of GPU devices in a way that is consistent with the abstraction and that can be implemented over all the back-ends. It is straightforward to allocate GPU resources when multiple GPUs run on a single machine, by means of the GPU API, however batch queuing systems, message passing infrastructures and the operating system are unaware of GPUs. As batch queuing systems are not GPU aware, one simple solution to the allocation of resources

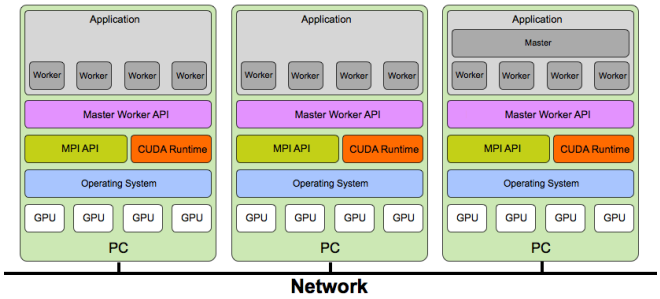

Fig. 1. Boss/Worker abstraction

is to assume that there is a one-to-one matching between nodes and GPUs so that one may rely on the existence of a GPU on each node. However we address heterogeneous clusters where some of the machines have one or more GPUs. The batch queuing system has no awareness of the GPUs so we let the system decide what nodes to allocate (this can be controlled configuring the batch queuing system), then the B/W software implements a handshaking mechanism that makes the Boss aware of the computing resources of each worker. When the application is launched, all the workers attempt to use the GPU API (*cudaGetDeviceCount* and *cudaGetDeviceProperties*) to retrieve the properties of the installed GPUs and communicate to the Boss a list of the GPUs installed on the local machine and their characteristics. The Boss then uses MPI node IDs (process IDs for the OS Pipe back-end) and GPU device number to figure out which nodes might conflict the allocation of a GPU on the same machine and assigns uniquely one GPU to each Worker; the Worker allocates the device that it has been assigned so each Worker is associated at most to one GPU. In order for all the GPUs to be in use it is necessary to configure the queuing system to create, on each machine, at least one node per GPU.

The Boss assigns a unique *WorkerID* to each Worker and stores information about whether the Worker is GPU capable and what is the type of GPU and its compute capability ($CUDA\_1.1$, $CUDA\_2.0$, ..). The programmer defines the Boss and the Worker functionalities by sub-classing two basic Boss and Worker classes. Each method of the Worker class may be executed by the Boss simultaneously on several Workers. A function *list_workers(compute_capability)* allows the Boss to obtain a list of Workers with a specific compute capability; the parameter to this function is a string that allows the Boss to query a list of all Workers that support GPU acceleration, or a list of the Workers that have a compute capability equal, greater or smaller then a specific value (for example =CUDA_2.0 lists all the Workers that have a CUDA device with compute capability exactly equal to 2.0).

The Boss has a *main* method that is executed when the Boss is first started by *M.engage()*. The programmer overrides the *main* method of the Boss to define the execution of tasks by the Workers.

In order to achieve automatic load balancing and to hide communication latency, the B/W implements and asynchronous API. The function *do(task_name, worker_ids, parameters)* determines buffering of function name, parameters and worker_ids and returns immediately a job ID to the Boss. In the background the function name and data objects are sent to the Workers. The second parameter to function *do* may also be a string that specifies compute capability; in this case the task is dispatched to any worker that has the requested compute capability. An idle Worker is chosen if available, otherwise the Worker with least number of tasks awaiting for execution is chosen. This mechanism is identical to the classical *task bag* [9][10] if the compute capability is set to *ANY*, but it allows to route tasks to nodes with GPU acceleration or specific compute capability.

If the Worker is busy, the function name, job ID and parameters are buffered on the Worker side. When the Worker completes the execution of a job, it notifies the Boss and pops a new job from the local buffer. The Boss disposes of two APIs to query the status and result of a job: the asynchronous interface returns *None* if the job (identified by its job ID) has not been completed, it returns the result otherwise; the synchronous interface, that blocks until the result is ready, is more efficient then polling the asynchronous interface when the Boss wants to synchronize with a specific job outcome. A *sync(jobIDs)* function is provided to wait until all jobs with given job IDs are completed or until all jobs are completed if the argument is 0.

The Workers are shut down explicitly by the Boss by means of a *poison pill* when the Boss shuts down. Before terminating, the Worker process deallocates the GPU device.

3 MPI Backend

The back-end for communication and data exchange can be implemented with a number of technologies, such as MPI, OS Pipes, Condor, BOINC, Tuple Space, addressing different cluster configurations. This allows us to run the same unmodified software on a single machine, on a cluster, or on a worldwide grid for cloud computing.

The only implementation of the back-end currently implemented is based on MPI-2. MPI and its batch queuing system provide the means to launch multiple processes on the nodes of a cluster and to send and receive messages in a synchronous and an asynchronous manner. The network layer is abstracted by assigning a node ID to each node. Though a more efficient beck-end might be designed to exploit shared memory when communicating amongst multiple processes on a single machine, MPI can create multiple nodes on the same machine and the OpenMPI implementation of MPI addresses the issue of efficient data transfer amongst processes.

In the B/W model, the Boss communicates with the Workers and the Workers send results back to the Boss, however there is no communication between Workers directly; each Worker has a *WorkerID* that is mapped, in the MPI back-end, to a node ID.

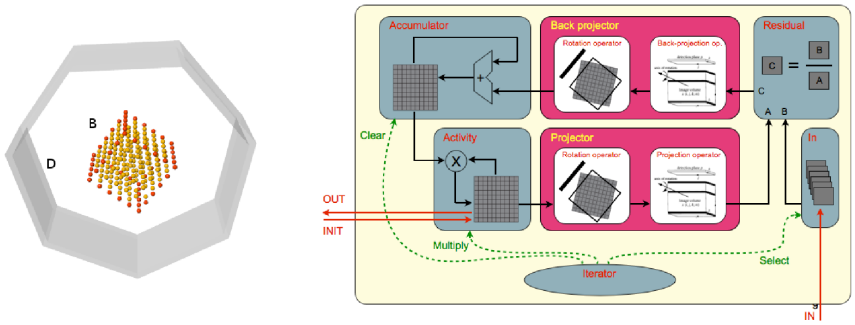

Fig. 2. Scheme of the SPECT Imaging system (left). Rotation Based MLEM Tomographic Reconstruction for Emission Tomography (right)

In order to implement the task bag with double queue and asynchronous behaviour, we use the asynchronous functions from the MPI API: asynchronous *send* to send messages from the Boss to the Workers and back and *iProbe* and *recv* to asynchronously receive the messages. We use the *tag* of the MPI message to define three message types: *JOB*, *RESULT*, *COMMAND*; the content of the message is always a dictionary, that has specific fields for the three types of messages. The JOB message has a *function* key, a *parameters* key, and a *jobid* key; *function* is a string that specifies the name of the function, *parameters* is a list object that lists all the parameters of the function (each with its own object type) and *jobid* is the integer ID that the Boss assigned to the job. The RESULT message, similarly, has a *result* key (list object) and a *jobid* key (integer). The COMMAND message specifies a *type* key (string object) and a *content* key (list object).

The back-end uses these three message types (tags) to send job requests, results and messages for handshaking and shutdown (*poison pill*).

4 Applications and Performance

We present the implementation of an application for Emission Tomographic Reconstruction. Through this application we show how the B/W programming model applies to a genuine problem and we evaluate the benefit that can be obtained by distributing the execution on multiple CPUs and GPUs by means of the B/W framework.

4.1 Tomographic reconstruction

Iterative stochastic algorithms for Tomographic Reconstruction in Emission Tomography (ET) provide high image quality, at the cost of high computational

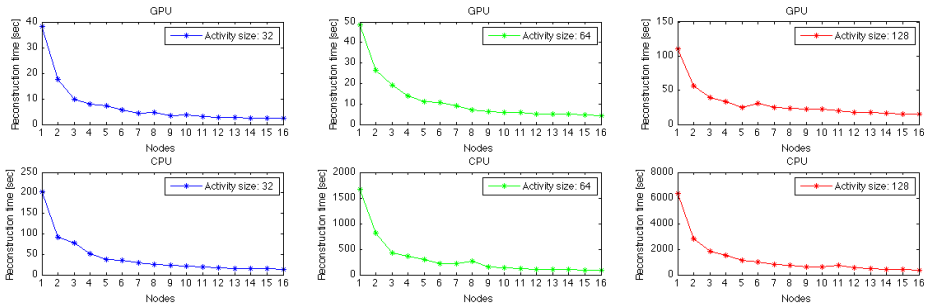

Fig. 3. Performance of MLEM Tomographic Reconstruction with the Boss/Worker framework for activity of size 32^3,64^3 and 128^3 voxels. Top figures show execution time for reconstruction on multiple GPUs; the figures in the bottom show execution time for reconstruction on multiple CPUs. The given time is for 60 iterations of MLEM and 180 projection angles.

complexity. We implemented a GPU accelerated Maximum Likelihood Expectation Maximization (MLEM) algorithm for ET reconstruction [11] and distributed its execution on multiple GPUs by means of the B/W framework.

The unknown 3-D activity $\hat{\lambda}_b$ (figure 2), defined on voxel space B, determines a projection image n_d in camera space D, where the projection space includes the pixels of the planar Gamma Camera at each position of the camera, as it rotates at regular steps during the scan; the imaging system being described by the transition matrix $p_{b,d}$.

The MLEM algorithm iteratively updates an estimate of the radio-tracer activity, with increasing likelihood of the estimation being the one that determined the observed data [11]:

$$\hat{\lambda}_b^{new} = \hat{\lambda}_b^{old} \frac{1}{\sum_{d=1}^{D} p_{bd}} \sum_{d=1}^{D} \frac{n_d \; p_{b,d}}{\sum_{b'=1}^{B} \hat{\lambda}_{b'}^{old} \; p_{b',d}}, \quad b = 1, .., B \tag{1}$$

A GPU accelerated projection/backprojection algorithm has been written in C/CUDA and is described in detail in [12]. The projection/backprojection function, which takes as input $\hat{\lambda}_b^{old}$, a list of cameras (their position and size), a point spread function, input projection data from each camera, and produces the backprojection (right of $\hat{\lambda}_b^{old}$ (1)), has been automatically wrapped with the wrapper generator based on SWIG for use with the B/W programming interface.

As can be observed from the MLEM formula in (1), the projection and backprojection can be calculated for any subgroup of D and the resulting backprojections need to be reduced by a sum operation in order to compute the new estimate of $\hat{\lambda}$. We exploit this task concurrency to execute the projection/backprojection on multiple GPUs buy letting each GPU compute the backprojection from a single camera (there are 180 cameras). The reduce operation is performed by the Boss for simplicity and the synchronous interface allows reduced memory

footprint as one projection is loaded from the workers at the time. The Python listing for the MLEM reconstruction is reported below.

```
from PyMW import Boss, Worker, Backend_MPI
from Spect import projector, save_activity, uniform_activity,\
create_psf, create_cameras, create_test_data, subset_cameras
N = 64;
n_cameras = 180;
out_name = 'activity.nii'

class Spect_Worker(Worker)
  def project(activity, psf, cameras):
    return projector.project(activity, psf, cameras)

class Spect_Boss(Boss)
  def reconstruct(self, sinogram, psf, cameras,activity):
    #use only nodes with GPU for projection
    workersIDs = self.interface.list_workers('GPU')
    id_list = []
    for i in range(len(workerIDs)):
      cameras_w = subset_cameras(cameras,len(workerIDs),i)
      #add job to the task bag
      id = self.interface.do('project',workerIDs[i],activity,\
                                                    psf,cameras_w)
      id_list.append(id)
    #reduction:
    for id in id_list:
      activity += self.interface.get_result(id)
    return activity

  def main(self):
    sinogram = create_test_data(N,'brain','poisson')
    psf      = create_psf(N,'ddpsf')
    cameras  = create_cameras(n_cameras,180)
    activity = uniform_activity(N)
    activity = self.reconstruct(sinogram,psf,cameras,activity)
    save_activity(activity,out_name)

if __name__ == "__main__":
  B = Backend_MPI()
  W = Spect_Worker(B)
  M = Spect_Boss(B)
  M.engage()
  W.engage()
```

The code has been tested for reconstruction on a small cluster of 16 computers, each equipped with a CPU Intel Core 2 Duo E6700 and a GPU NVIDIA GTX 285 with 1Gb of DDR3; the PCs are connected by means of 1Gbps Ethernet. Each machine is equipped with Ubuntu-9.4 with NVIDIA drivers 195.36.1 and they share a network drive where we have installed Python-2.5, Numpy-1.3.0 and MPI4Py built with OpenMPI-1.4.1. We launch MPI by means of *ssh*. In figure 3 we report the execution time registered for reconstruction until convergence (60 MLEM iterations) with 180 projections and three different sizes of the activity: 32^3, 64^3, 128^3 voxels. The three images on the top represent the reconstruction time for the GPU implementation while the three in the bottom are for the CPU implementation. These images show the speed up of approximately 100-fold of the GPU implementation over the implementation for CPU, when executed on a single core (no multi-threading support was implemented for the CPU version). When the size of the data is small, the improvement is less marked because of

the time it takes to launch the kernels on the GPUs (tough the recently released Fermi devices claim a 10-fold improvement in kernel scheduling time).

Scaling the execution on multiple GPUs gives an immediate benefit, but the gain in performance drops when using more than 8 GPUs. This is due to the latency when transferring the activity to the the Workers and back, in fact there is no other task subsequent to projection/backprojection, so after each projection/backprojection task all the workers need to synchronize to perform the reduction and produce the new estimation of the activity. When the algorithm is distributed on 8 GPUs we obtain a speed up of about 4, while the speed up for 8 CPU is of about 5 times. However it is remarkable that it is not possible to achieve the performance of a single GPU with a cluster of CPUs, due to communication latency predominating the computation.

5 Conclusion

The aim of our work is to provide a Boss/Worker abstraction that allows to distribute code on multiple machines in number of computing environments as diverse as possible. We have reviewed the Boss/Worker programming model showing its applicability and limitations. The B/W abstraction provides an effective way to distribute execution of code when the problem presents a high degree of task concurrency. In this case a B/W programming model is appealing as it provides abstraction from the communication protocol between processes and because it provides automatic load balancing and hides communication latency. We have presented the implementation of a Python based B/W framework that allows to distribute an application, unmodified, on a number of diverse computing environments; particularly we have addressed distribution of code in a multi-GPU environment. We have addressed a number of issues involved in distributing workload to multiple GPUs and shown that the B/W abstraction is well suited to multi-GPU programming.

Finally we have tested our B/W framework by implementing an algorithm for Tomographic Reconstruction in Emission Tomography. This application shows that the framework allows to distribute execution on multiple GPUs with little effort for the programmer. However for this application, the scalability is limited to a maximum of 8 to 16 GPUs because of the low computational intensity of the macroscopic tasks that can be extracted. Nonetheless, a speed up of a factor 4 is achieved by distributing the execution of the algorithm on 6/8 GPUs. Because of the intrinsic low computational intensity of the problem taken into consideration, the reconstruction algorithm does not scale well neither on CPU nor on GPU, but the GPU implementation provides a speed up of up to 100 times over the CPU implementation. The B/W framework introduces another speed up of a factor 4. The B/W programming model proves to simplify the distribution of code to multiple machines and GPUs, and its unified interface provides the means to distribute code on clusters of diverse sizes and configurations. It can be applied to scale a GPU based application to run on a small GPU cluster, or to scale an algorithm that presents sufficient task parallelism on a bigger cluster.

References

1. Stone, J., Phillips, J., Freddolino, P., Hardy, D., Trabuco, L., Schulten, K.: Accelerating molecular modeling applications with graphics processors. **28** (2007) 2618–2640
2. Ufimtsev, I., Martinez, T.: Quantum chemistry on grapical processing units. 1. trategies for two-electron integral evaluation. **4**(2) (2008) 222–231
3. Pedemonte, S., Gola, A., Abba, A., Fiorini, C.: Optimum real-time reconstruction of gamma events for high resolution Anger camera. (Oct. 2009)
4. Schellmann, M., Gorlatch, S., Meilander, D., Kosters, T., Schafers, K., Wubbeling, F., Burger, M.: Parallel medical image reconstruction: From graphics processors to grids. PaCT 2009 **LNCS**(5698) (2009) 457–473
5. Scherl, H., Hoppe, S., Kowarschik, M.: Design and implementation of the software architecture for a 3-d reconstruction system in medical imaging. In: ICSE, Leipzig, Germany (2008)
6. Phillips, J., Stone, J., Schulten, K.: Adapting a message-driven parallel application to gpu-accelerated clusters. SC2008 (2008)
7. Dongarra, J., Pineau, J., Robert, Y., Shi, Z., Vivien, F.: Revisiting matrix product on master-worker platforms. International Journal of Foundations of Computer Science **19**(6) (2008) 1317–1336
8. Dongarra, J., Pineau, J., Robert, Y., Vivien, F.: Matrix product on heterogeneous master-worker platforms. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, Salt Lake City, UT, USA (2008) 53–62
9. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison Wesley (2005)
10. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
11. Shepp, L., Vardi, Y.: Maximum likelihood reconstruction for emission tomography. IEEE Trans. on Medical Imaging **1**(2) (1982) 113–22
12. Pedemonte, S., Bousse, A., Erlandsson, K., Modat, M., Arridge, S., Hutton, B., Ourselin, S.: GPU Accelerated Rotation-Based Emission Tomography Reconstruction. (Nov 2010) 2657–2661